Requirements:

- A common framework for all systems → easy maintainance of the whole system
- Stability → easy to program, clear structure
- Robustness → simple and well defined error handling

Pre-requisites:

- Our subsystems execute loops, for example:

  - comparing actual values with nominal values and send new command values
  - reading actual values and writing them to a file
  - wait for something (new command, a nominal value to be reached, …)
  - *Loops* with just one step (initialize hardware, open file, ...)

  → Can be split into single steps, loop control is taken by the framework
  → Each kind of loop (action) which is executed is called “State of the system”
  → The transition between two states are instantaneous, but well defined,
  i.e. outside the execution-step
    - A table defines the allowed transitions
    - Transitions can be triggered externally (e.g. commands)
    - Transitions can be automatic
      (e.g. when a file got opened the state is changed to “writing”)
    - Errors can easily be propagated and handled

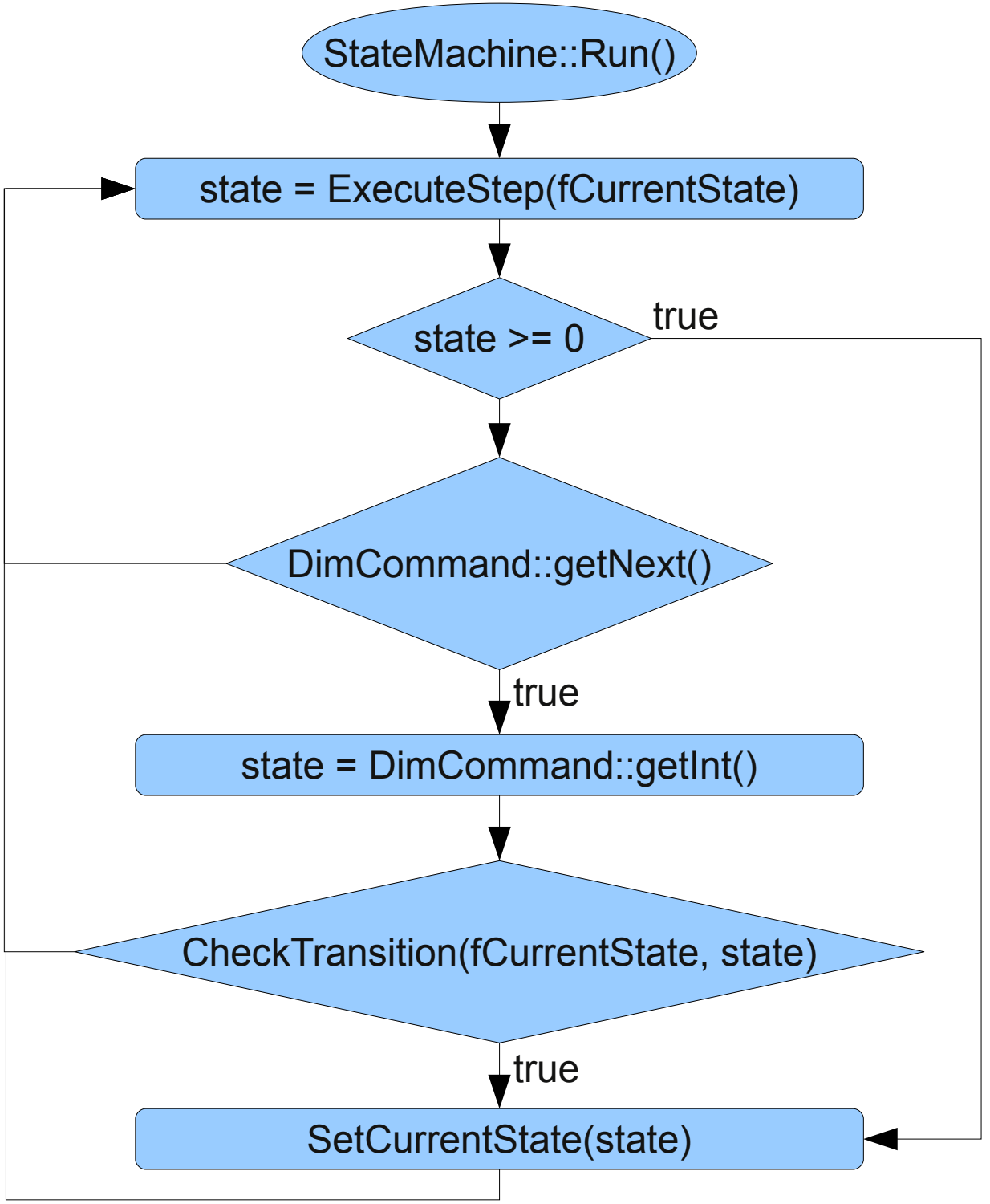
The framework will deliver the loop control and all the necessary bi-directional communication

The subsystem programmer has to implement:

- newstate=ExecuteStep(state):
  - The function which will execute an action (“step”) depending on the current state (This should never include its own loop)
  - Returning a new state if a new state is necessary
  - Whenever the function is left the status of the system must be well defined
- CheckTransition(oldstate, newstate)
  - Returns if a transition from one state into another state is allowed
  - Transitions which are issued by ExecuteStep are not checked and the responsibility of the programmer.

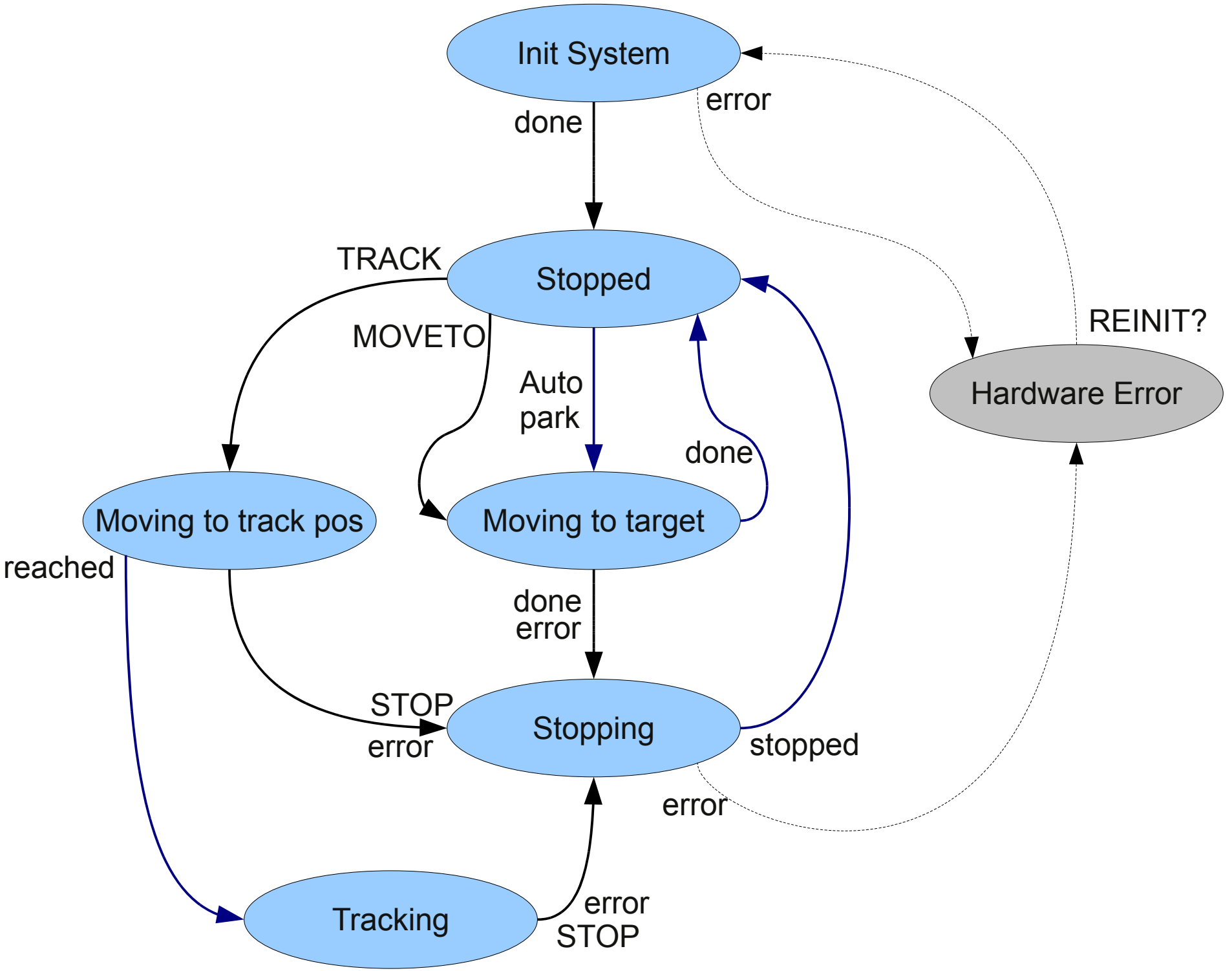
The framework will take care of

- Only changing the state accoring to the allowed transitions
- Making sure that for each new state which was applied, ExecuteState is called at least once
- Automatic (returned) state transition have priority over external state transitions
- The framework will handle the corresponding communication and will buils a command queue


StateMachine::Run()
state = ExecuteStep(fCurrentState)
state >= 0
true
DimCommand::getNext()
true
state = DimCommand::getInt()
CheckTransition(fCurrentState, state)
true
SetCurrentState(state)

Advantage of a state machine:

- It forces the programmer to break down his program into well defined small parts.
  - You are forced to well define the entry points and the exit points of your step
- The small parts of the program are easy to document and therefore easy to maintain.
- The whole program can be easily drawn graphically which improves maintainability and error search.
- It is easy to get a stable program because the program and hardware status is well defined.
- It is easy to get the whole system stable because any reaction is well defined.
- Everything is excuted sequentially – no threading issues.
- By taking a local copy of the command values all threading issues can easily be avoided (maybe this can be part of the framework)
- Due to the well defined states, error handling becomes easy and straight forward which is a major improvement for robustness and stability! (No need to exit the program anymore somewhere since errors can easily be propagated and if necessary the program can be exited cleanly)


Init System
done
error
TRACK
Stopped
MOVETO
Auto
park
REINIT?
Hardware Error
done
Moving to track pos
Moving to target
reached
done
error
STOP
error
Stopping
stopped
error
Tracking
error
STOP